Fortify Security Report

1/31/23

Demo User

Executive Summary

Issues Overview

On 31 Jan 2023, a source code review was performed over the FortifyMuleSoftDemo code base. 29 files, 35 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 22 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
High	16
Critical	3
Low	3

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Users/klee/source/repos/FortifyMuleSoftDemo
Number of Files: 29
Lines of Code: 35
Build Label: SNAPSHOT

Scan Information

Scan time: 00:18
SCA Engine version: 22.2.0.0130
Machine Name: GBklee02
Username running scan: klee

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
com.microfocus.example.StockService.main

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low

Audit Guide Summary

Audit guide not enabled

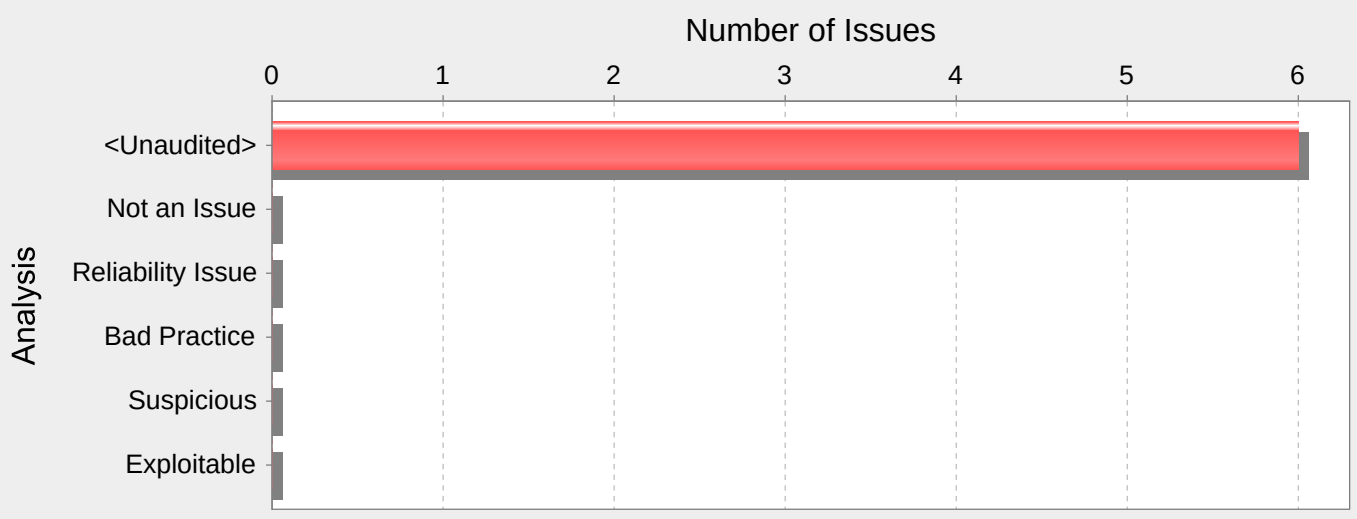
Results Outline

Overall number of results

The scan found 22 issues.

Vulnerability Examples by Category

Category: Password Management: Hardcoded Password (6 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is difficult to remedy.

Explanation:

Never hardcode passwords. Not only does it expose the password to all of the project's developers, it also makes fixing the problem extremely difficult. After the code is in production, a program patch is probably the only way to change the password. If the account the password protects is compromised, the system owners must choose between security and availability.

Example: The following YAML uses a hardcoded password:

...
credential_settings:
username: scott
password: tiger
...

This configuration may be valid, but anyone who has access to the configuration will have access to the password. After the program is released, changing the default user account "scott" with a password of "tiger" is difficult. Anyone with access to this information can use it to break into the system.

Recommendations:

Never hardcode password. Passwords should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

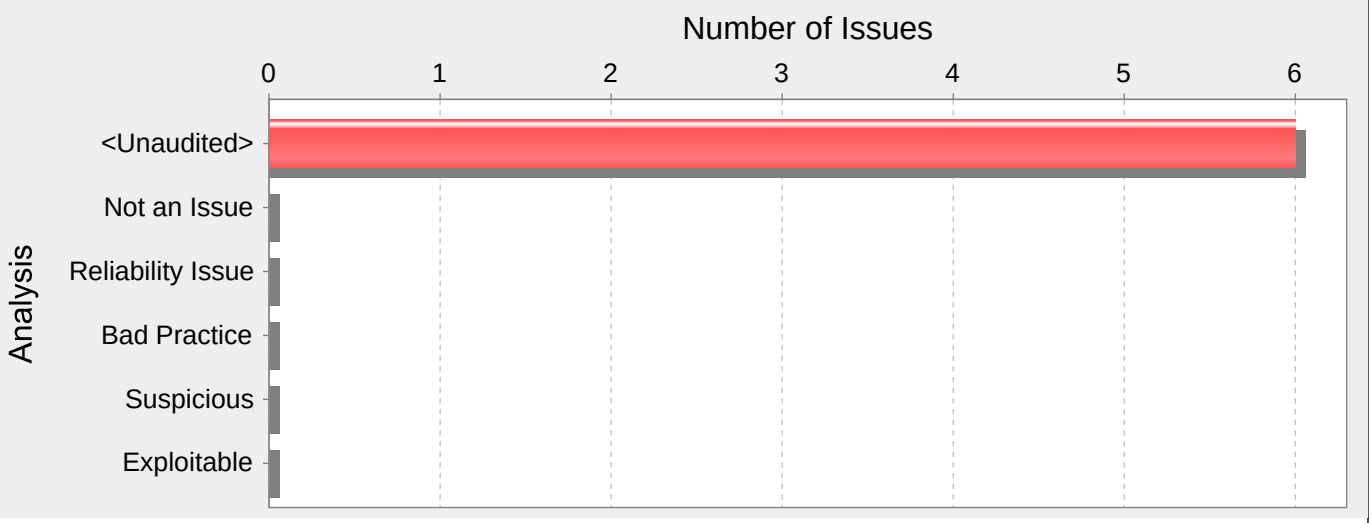
config-dev.yaml, line 22 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is difficult to remedy.		

Sink: config-dev.yaml:22 ConfigPair()

```
20    host: "localhost"
21    port: "3306"
22    user: "fortifymule"
23    password: "fortifymule"
24
```

Category: Password Management: Password in Configuration File (6 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is difficult to remedy.

Explanation:

Never hardcode passwords. Not only does it expose the password to all of the project's developers, it also makes fixing the problem extremely difficult. After the code is in production, a program patch is probably the only way to change the password. If the account the password protects is compromised, the system owners must choose between security and availability.

Example: The following XML uses a hardcoded password:

...
<user>
<username>scott</username>
<secretPassword>tiger</secretPassword>
</user>
...

This configuration may be valid, but anyone who has access to the configuration will have access to the password. After the program is released, changing the default user account "scott" with a password of "tiger" is difficult. Anyone with access to this information can use it to break into the system.

Recommendations:

Never hardcode password. Passwords should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

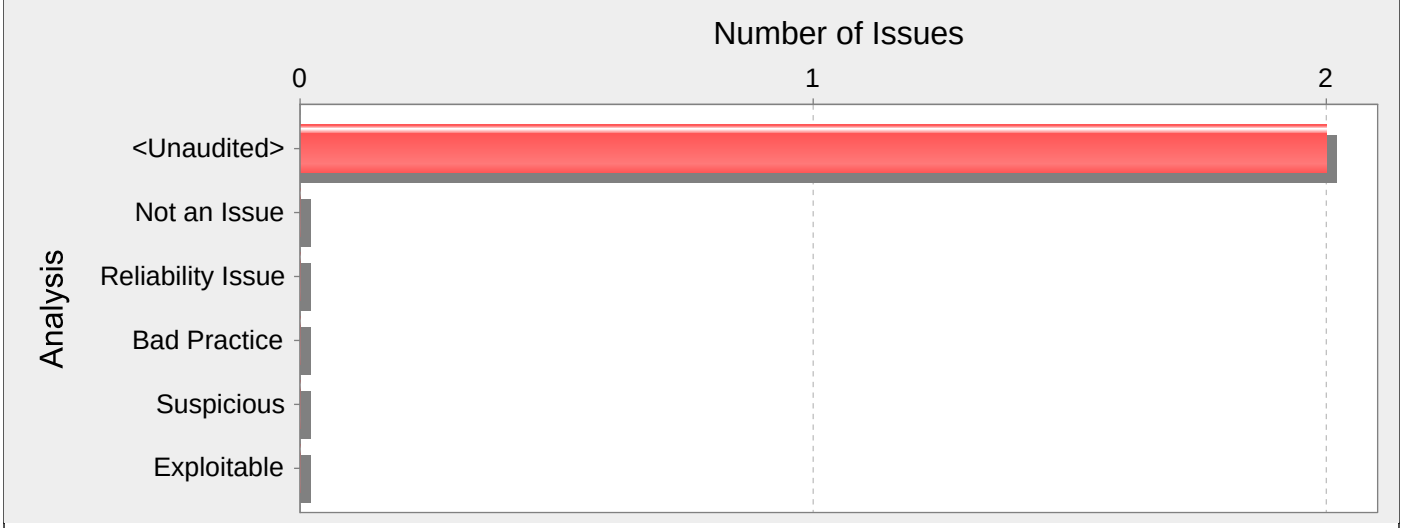
store-api.xml, line 44 (Password Management: Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Hardcoded passwords can compromise system security in a way that is difficult to remedy.		

Sink: store-api.xml:44 null()

```
42    <db:my-sql-connection host="localhost"
43      port="3306" user="fortifymule" password******
44      database="fortifymule">
45      <db:connection-properties>
46       <db:connection-property key="useSSL" value="false" />
```

Category: Insecure Randomness (2 Issues)



Abstract:

The random number generator implemented by nextInt() cannot withstand a cryptographic attack.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and form an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between the generated random value and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
String GenerateReceiptURL(String baseUrl) {
Random ranGen = new Random();
ranGen.setSeed((new Date()).getTime());
return (baseUrl + ranGen.nextInt(400000000) + ".html");
}
```

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Since Random.nextInt() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Do not use values such as the current time because it offers only negligible entropy.)

The Java language provides a cryptographic PRNG in java.security.SecureRandom. As is the case with other algorithm-based classes in java.security, SecureRandom provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a SecureRandom object using SecureRandom.getInstance(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a SecureRandom object. If it is unavailable or if you do not specify a particular implementation, then you are given a SecureRandom implementation selected by the system.

Sun provides a single SecureRandom implementation with the Java distribution named SHA1PRNG, which Sun describes as computing:

"The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]."

However, the specifics of the Sun implementation of the SHA1PRNG algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

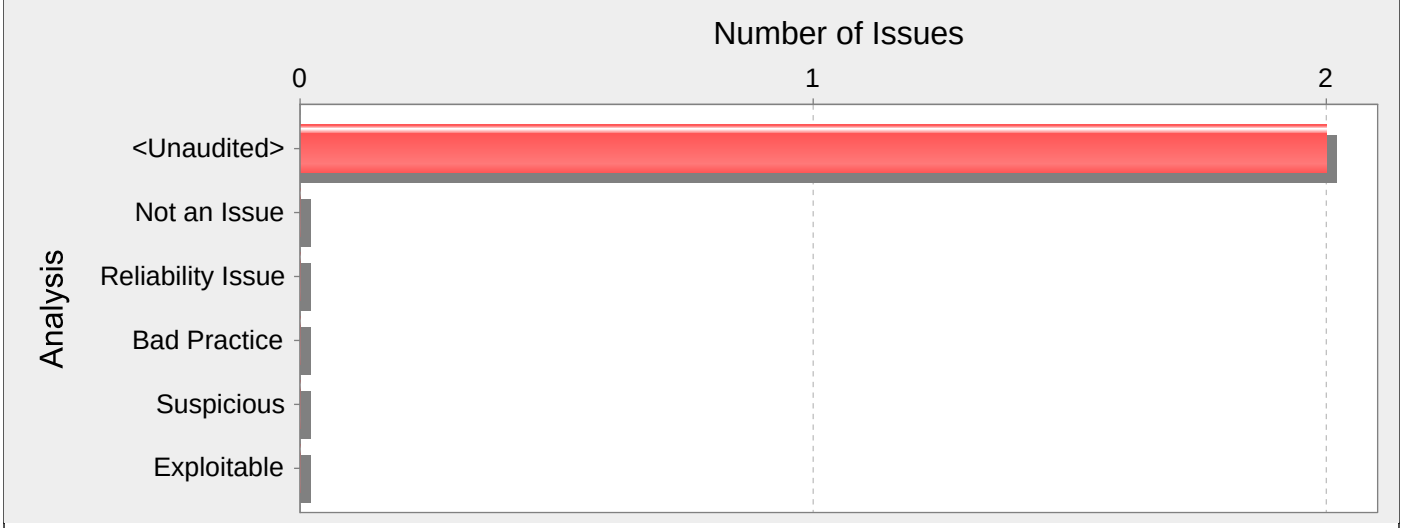
StockService.java, line 18 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by nextInt() cannot withstand a cryptographic attack.		

Sink: StockService.java:18 nextInt()

```
16                System.out.println("Product id: " + productId);
17              Random rand = new Random(); //instance of random class
18                int stock_level = rand.nextInt(MAX_STOCK_LEVEL);
19                System.out.println("Stock level: " + stock_level);
20                return stock_level;
```

Category: Key Management: Hardcoded Encryption Key (2 Issues)



Abstract:

Hardcoded encryption keys can compromise security in a way that is not easy to remedy.

Explanation:

Never hardcode an encryption key because it makes the encryption key visible to all of the project's developers, and makes fixing the problem extremely difficult. Changing the encryption key after the code is in production requires a software patch. If the account that the encryption key protects is compromised, the organization must choose between security and system availability.

Example 1:

The Public-Key Cryptography Standards #12 KeyStore is a common cryptographic bundle file format, which commonly contains SSL/TLS certificates and corresponding private keys. It uses file name extensions .pkcs12, .p12, and pfx.

Anyone with access to the code can see the encryption key. After the application has shipped, there is no way to change the encryption key unless the program is patched. An employee with access to this information can use it to break into the system. Any attacker with access to the application executable can extract the encryption key value.

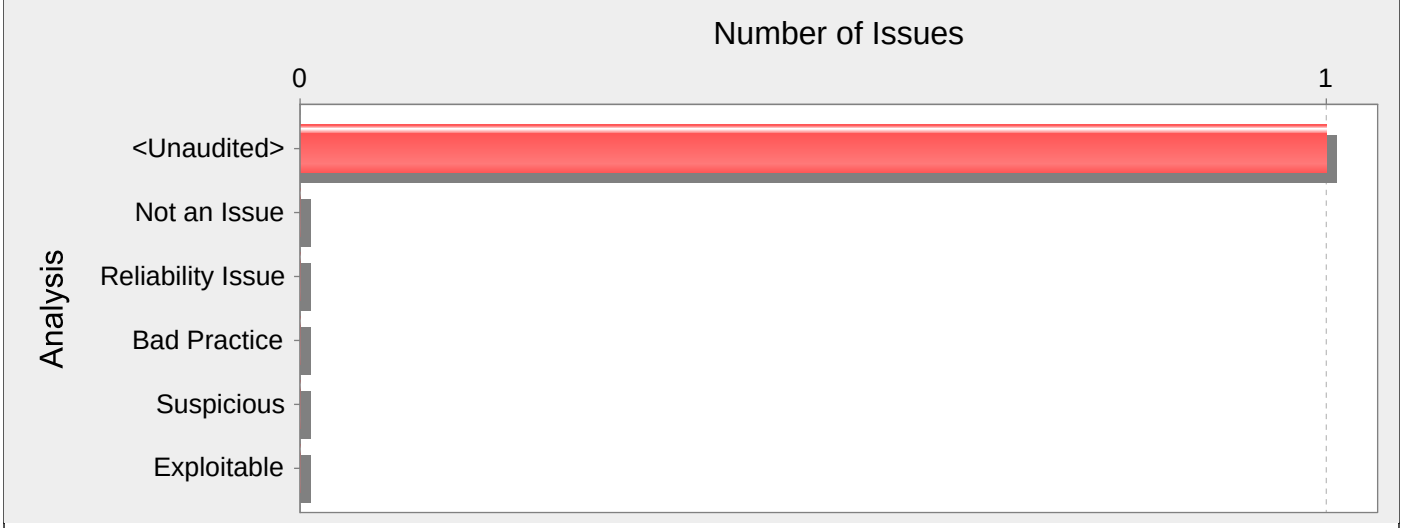
Recommendations:

Never check in encryption keys to your source control system, and never hardcode them. Always obfuscate and manage encryption keys in an external source. Storing encryption keys in plain text anywhere on the system enables anyone with sufficient permissions to read and potentially misuse the encryption key.

server-dev-keystore.p12, line 0 (Key Management: Hardcoded Encryption Key)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded encryption keys can compromise security in a way that is not easy to remedy.		
Sink:	server-dev-keystore.p12:0 null()		
`-2`	`0,`		
`-1`	`µ^B^A^C0,`		

Category: Mule Misconfiguration: Hardcoded Password (1 Issues)



Abstract:

On line 44 of store-api.xml, an Anypoint Database Connector configuration contains a plain text password.

Explanation:

Hardcoding a password allows all project developers to view the password. It also makes fixing the problem difficult. After the code is in production, no one can change the password without patching the software. If the account protected by the password is compromised, the system's owners must choose between security and availability.

Example 1: The following Mule configuration stores a plain text password for a MySQL database connector.

```
<db:config ... >
<db:my-sql-connection ... password="hardcoded-password" ... >
...
</db:my-sql-connection>
</db:config>
```

Recommendations:

Use the Mule runtime engine to retrieve sensitive configuration properties from a secure source such as a Secure Configuration Properties file and a credentials vault.

Example 2: The following example fixes the problem in Example 1. The environment-dependent property ${mule.env} enables the operations team to maintain a restricted access file (${mule.env}.properties) that stores the production configuration properties. The Mule runtime engine dynamically retrieves the password (${dbpassword}) from the properties file and assigns it to the password property of a MySQL database connection.

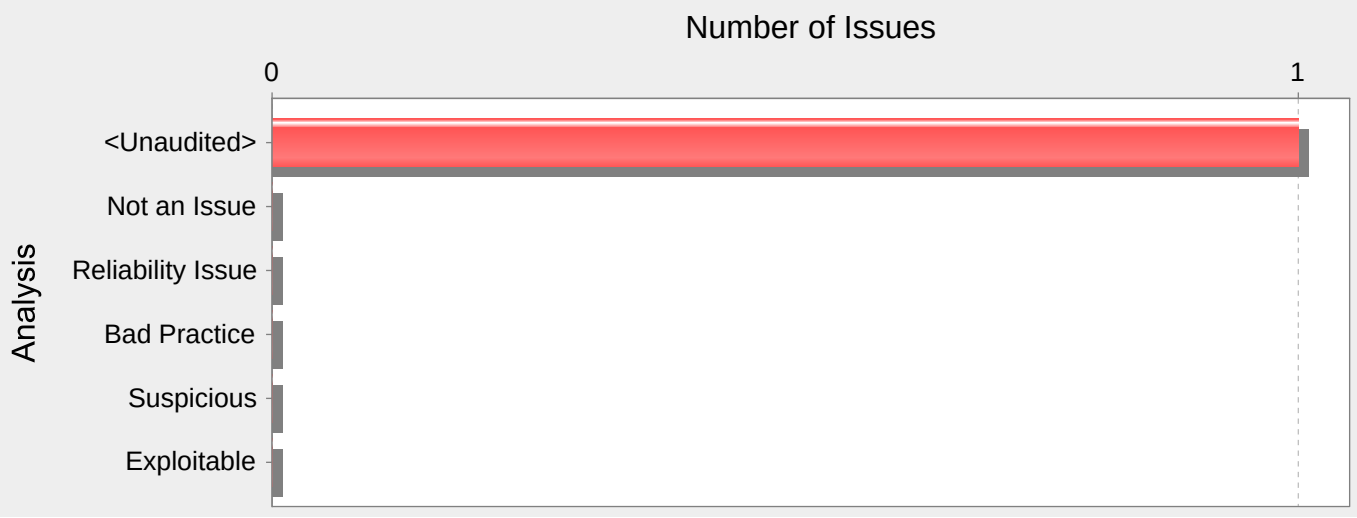
```
<global-property name="mule.env" value="dev"/>
<configuration-properties file="${mule.env}.properties"/>
...
<db:config ... >
<db:my-sql-connection ... password="${dbpassword}" ... >
...
</db:my-sql-connection>
</db:config>
```

store-api.xml, line 44 (Mule Misconfiguration: Hardcoded Password)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Environment		
Abstract:	On line 44 of store-api.xml, an Anypoint Database Connector configuration contains a plain text password.		
Sink:	store-api.xml:44 null()		

```
42    <db:my-sql-connection host="localhost"
43      port="3306" user="fortifymule" password=******
44      database="fortifymule">
45      <db:connection-properties>
46       <db:connection-property key="useSSL" value="false" />
```

Category: Mule Misconfiguration: Insecure Database Transport (1 Issues)



Abstract:

In store-api.xml on line 44, a database connection is created without enforcing encryption.

Explanation:

The Mule Database Connector should force encrypted transport and verify database server certificates. If relevant database connection properties are left unspecified or misconfigured, the resulting connections expose the data to unauthorized access, tampering, and potential theft.

Use any of the following combinations of the Database Connector connection properties to secure MySQL Server database connections:

- sslMode = VERIFY_CA
- sslMode = VERIFY_IDENTITY
- (UseSSL = true) and (requireSSL = true) and (verifyServerCertificate = true)

Example 1: The following Mule configuration does not define any of the recommended connection properties. The resulting database connections are insecure.

```
...
<db:config name="db_config_mysql">
<db:my-sql-connection host="db.example.com" user="foobar" password="${password}" databaseName="mysqldemo"/>
</db:config>
...
```

Recommendations:

Configure database connection properties to force encrypted transport and verify server certificates.

Example 2: The following example fixes the problem in Example 1 by setting the sslMode connection property to VERIFY_CA.

```
...
<db:config name="db_config_mysql">
<db:my-sql-connection host="db.example.com" user="foobar" password="${password}" databaseName="mysqldemo">
<db:connection-properties>
<db:connection-property key="sslMode" value="VERIFY_CA"/>
</db:connection-properties>
</db:my-sql-connection>
</db:config>
...
```

Tips:

1. Older JDBC driver and database server versions might not support specific connection properties.

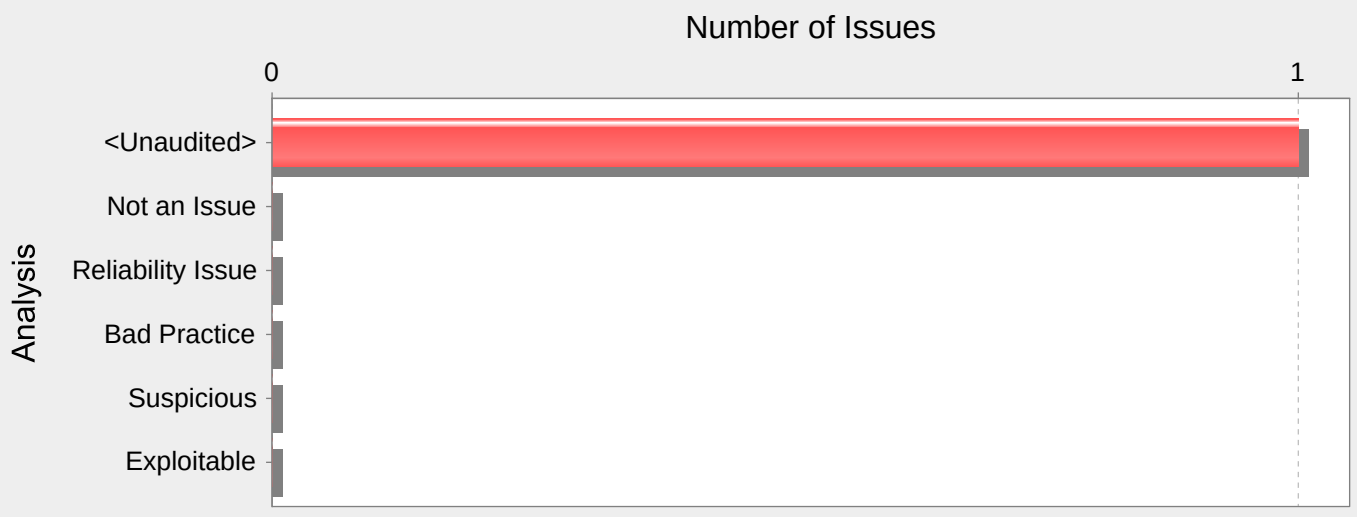
store-api.xml, line 44 (Mule Misconfiguration: Insecure Database Transport)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Environment		
Abstract:	In store-api.xml on line 44, a database connection is created without enforcing encryption.		

Sink: store-api.xml:44 null()

```
42    <db:my-sql-connection host="localhost"
43     port="3306" user="fortifymule" password******
44     database="fortifymule">
45     <db:connection-properties>
46      <db:connection-property key="useSSL" value="false" />
```

Category: Mule Misconfiguration: Server Identity Verification Disabled (1 Issues)



Abstract:

On line 42 of mule-domain-config.xml, the Mule configuration specifies a TLS connection without the server certificate validation check.

Explanation:

Certification verification is essential to confirm the counterpart's identity for secure communication. A tls:context element defines a set of TLS connection configurations. Among the configurations, the tls:trust-store element specifies a file that contains certificates from trusted Certificate Authorities that a client uses to verify a certificate presented by a server. By default, the Mule runtime engine verifies the server certificate for every TLS connection.

However, if the value of the insecure attribute of the tls:trust-store element is true, server certificates are accepted without verification.

Example 1: The following Mule configuration sets the insecure attribute to true. As a result, the Mule runtime engine does not verify the server certificate of any connection with the TLS context named demoTlsContext. Such a connection is susceptible to a man-in-the-middle attack.

...
<tls:context name="demoTlsContext">
...
<tls:trust-store ... insecure="true" ... />
...
<tls:context/>
...

Recommendations:

In production environments, always verify TLS server certificates.

Example 2: The following example fixes the problem in Example 1. It forces certificate verification by setting the insecure attribute of the tls:trust-store element to false.

...
<tls:context name="demoTlsContext">
...
<tls:trust-store ... insecure="false" ... />
...
<tls:context/>
...

mule-domain-config.xml, line 42 (Mule Misconfiguration: Server Identity Verification Disabled)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Environment		
Abstract:	On line 42 of mule-domain-config.xml, the Mule configuration specifies a TLS connection without the server certificate validation check.		
Sink:	mule-domain-config.xml:42 null()		

```
40    <tls:context name="serverTlscontext"
41     enabledProtocols="TLSv1.2">
```

```
42                          <tls:trust-store type="jceks" insecure="true" />
43                          <tls:key-store type="pkcs12"
44                           path="ssl/server-dev-keystore.p12" alias="server"
```

Issue Count by Category

Issues by Category	
Password Management: Hardcoded Password	6
Password Management: Password in Configuration File	6
Insecure Randomness	2
Key Management: Hardcoded Encryption Key	2
Password Management: Password in Comment	2
J2EE Bad Practices: Leftover Debug Code	1
Mule Misconfiguration: Hardcoded Password	1
Mule Misconfiguration: Insecure Database Transport	1
Mule Misconfiguration: Server Identity Verification Disabled	1

Issue Breakdown by Analysis

Issues by Analysis